Writing your first Linux kernel module

Praktikum Kernel Programming

University of Hamburg
Scientific Computing
Winter semester 2015/2016

Konstantinos Chasapis
Konstantinos.chasapis@informatik.uni-hamburg.de

Outline

- Before you start
- Hello world module
- Build, load and unload
- User VS Kernel space programming

Before you start

- Define your module's goal
- Define your module behaviour
- Know your hardware specifications
  - If you are building a device driver you should have the manual
- Documentation
  - /usr/src/linux/Documentation
  - make { htmldocs | psdocs | pdfdocks | rtfdocks }
  - /usr/src/linux/Documentation/DocBook

Role of the device driver

- Software layer between application and device "black boxes"
  - Offer abstraction
    - Make hardware available to users
  - Hide complexity
    - User does not need to know their implementation
- Provide mechanism not policy
  - **Mechanism**
    - Providing the flexibility and the ability the device supports
  - **Policy**
    - Controlling how these capabilities are being used

Role of the device driver

- Policy-free characteristics
  - Synchronous and asynchronous operations
  - Exploit the full capabilities of the hardware
  - Often a client library is provided as well
    - Provides capabilities that do not need to be implemented inside the module

Outline

- Before you start
- Hello world module
- Build, load and unload
- User VS Kernel space programming

Hello world module

```c
/* header files */
#include <linux/module.h>
#include <linux/init.h>
/* the initialization function */
    static int __init hello_init(void) {
      printk( "Hello world !\n");
      return 0; /* success */
    }

    /* declares which function will
    be invoked when the module is
    loaded */
    module_init(hello_init);
```

```c
/* the shutdown function */
    static void __exit
    hello_exit(void) {
      printk("Goodbye,!\n");
    }

/* declares which function will be
    invoked when the module is
    removed */
    module_exit(hello_exit);
```

Initialization function

- Each module must use one
- Declared as static
- __init <name>
  - Use only at initialization
- __initdata
  - Mark initialization data
- Does not accept parameters
- Returns error code
- Kernel drops init function and data
  - Makes the memory available to the system

```
static int __init hello_init(void) {
    printk( "Hello world !\n");
    return 0; /* success */
  }

  module_init(hello_init);
```

Shutdown function

- Only if you need to unload the module
- Declared as static
- __exit <name>
  - only at shutdown
- module_exit(<name>)
- If not defined
  - Modules can not be unloaded
- The build in modules do not require shutdown

```
static void __exit hello_exit(void) {
      printk("Goodbye,!\n");
   }

module_exit(hello_exit);
```

printk

- Similar to printf but:
  - Prints to the kernel log file
  - Does not support all the formatting parameters
- Very expensive operation
  - Lots of printk's can significantly slow down the system
- Accepts loglevels
  - A hint to the kernel to decide if it should print the string to the log file
  - Default KERN_WARNING

printk - loglevels

- KERN_EMER
  - An emergency condition
- KERN_ALERT,
  - requires immediate attention
- KERN_CRIT
- KERN_ERR
- KERN_WARNING
- KERN_NOTICE
- KERN_INFO
- KERN_DEBUG

Module parameters

- Pass parameters to the module through
  - insmod
  - modprobe
- modprobe reads parameters thought
  - /etc/modprobe
- Read parameter value while module is loaded
  - cat sys/module/<mod_na>/parameters/<param_na>

Module parameters

- Parameter declaration
  - module_param(name, type, permission)
    - Permissions modes are as file access modes
    - Parameters types:
      - bool, inbool (inverted bool)
      - charp, string
      - int, long, short
      - uint, ulong, ushort
- Also accepts arrays parameters
  - module_param_array(name, type, nump, perm)

Error handling

- Failure may occur during initialization phase
  - memory allocation
  - device is busy
- continue or drop?
  - If we drop
    - undo any registration activities performed before
    - in case we fail to unregister the kernel goes into unstable mode
- Recovery is usually handle with the goto statement

Error handling

- Error number definitions at <linux/errno.h>
  - Return negative values -error code;
- #define EPERM 1 /* Operation not permitted */
- #define ENOENT 2 /* No such file or directory */
- #define EIO 5 /* I/O error */
- #define ENOEXEC 8 /* Exec format error */
- #define EAGAIN 11 /* Try again */
- #define ENOMEM 12 /* Out of memory */
- #define EACCES 13 /* Permission denied */
- #define ENOSYS 38 /* Function not implemented */
- #define ENOTEMPTY 39 /* Directory not empty */

Outline

- Before you start
- Hello world module
- Build, load and unload
- User VS Kernel space programming

Compile

- kbuild
  - the system that is used to compile kernel modules
  - /Documentation/kbuild/
- You must have a pre-build kernel with configuration and header files
- Many distributions have packages for the required files and tools
  - kernel-devel package for CentOS

Compile command

- make -C $KDIR M=$PWD [target]
  - $KDIR
    - the directory where the kernel source is located.
      - make will change the directory for the compile and will return after the compile
  - M=$PWD
    - Informs kbuild that an external module is being build.
    - The value of M is the absolute path the directory that contains the source code of the module

make command targets

- modules
  - The default target that can be ignored
- modules_install
  - Installs the external modules
  - The default location is /lib/modules/<kernel_release>/ extra/
- clean
  - remove all generated files in the module directory only
- help
  - list the available target for the external modules

kbuild file

- Contains the name of the module(s) being built, along with the requisite source files
  - obj-m := <m_name>.o
    - kbuild will build <m_name>.o from <m_name>.c
  - Then it will link it and will result in the kernel module <m_name>.ko
  - An additional line is needed to add more files
    - <module_name>-y := <src1>.o <src2>.o ....
  - Include files and directories
    - standard files using #include <file>
    - ccflags-y := -Iinclude_path

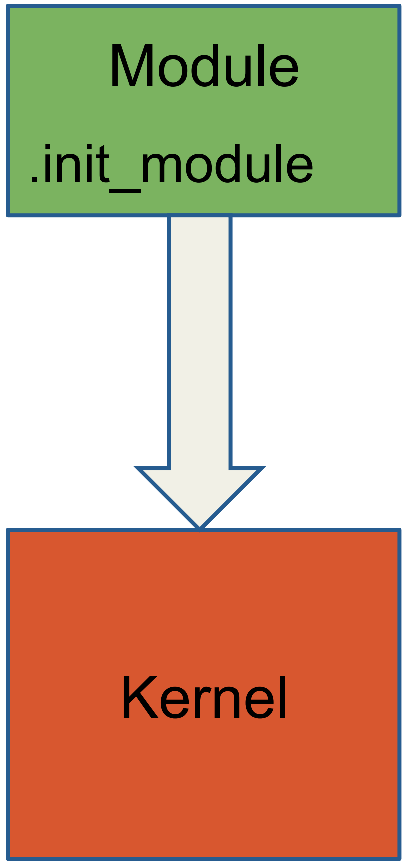
Module.symvers file

- Module versioning is enabled by the CONFIG_MODVERSIONS tag
- It is used as a simple Application Binary Interface (ABI) consistency check

- It contains a list of all exported symbols from a kernel build
  - /proc/kallsyms

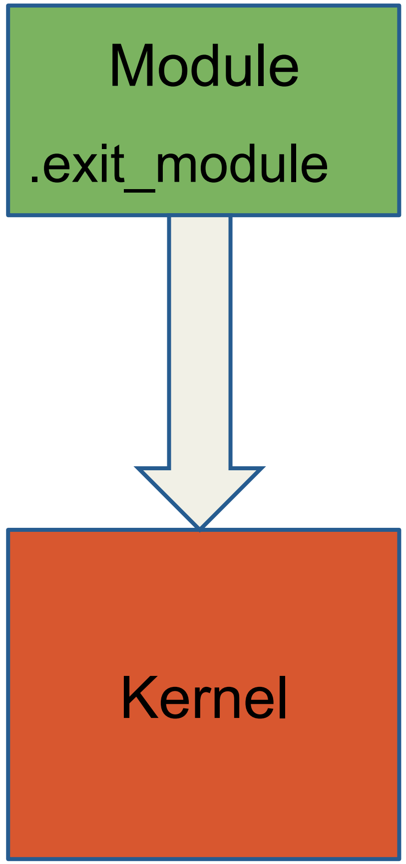
insmod (insert module)

- load the module into the kernel
  - triggers the execution of the module_init function
- Similar to the ld in user space
- Load the module code and data into the kernel memory
- Links any unresolved symbol in the module to the symbol table of the kernel
- Accepts command line arguments
  - Parameters to the kernel module
- Add an entry at /proc/modules
- For more details check kernel/module.c



rmmod (remove module)

- Removes/unloads the module from the kernel
- Must free memory and release recourse
- In case of failure the kernel still believes that the module is in use
- In case that rmmod fails the reboot process is required to clean the systems state



More tools

- lsmod (list modules)
  – List of the current loaded modules
- modprobe (similar to insmod)
  – Search for symbols that are not currently defined in the kernel
  – In case that there are then search for in kernel modules to find modules that contain these symbols
  – It loads these modules into the kernel
- depmod
  – Creates a dependency file, used by modprobe
- modinfo
  – Shows information about a Linux Kernel module

Version dependency

- Modules have to be recompiled for each version
  - data structures and function prototypes can changes from version to version
  - during compilation the module is linked against a file named vermagic.o
  - This file contains target kernel version, compiler version etc.
- In case that the module is compile against different kernel version
  - insmod: Invalid module format

Version dependency (cont.)

- Macros to define kernel version during compilation found in /linux/version.h
  - UTS_RELEASE, the version of this kernel tree
  - LINUX_VERSION_CODE, binary representation of the kernel version
  - KERNEL_VERSION(major, minor, release), build an inter version code

Kernel Symbol Table

- Kernel has already exported symbols
- Loaded modules can export new symbols
  - offer their functionality to other modules
- Stack modules on top of other modules
  - Reduce complexity of the modules
  - Add flexibility to choose modules depending on the specific hardware
- Macros to export new symbols
  - EXPORT_SYMBOL(name);
  - EXPORT_SYMBOL_GPL(make);
- Expand into specific variable declarations stored in the module executable file

dkms

- Dynamic Kernel Module Support
  - Framework that enables generating Linux kernel modules whose sources generally reside outside the kernel source tree
  - Used to automatically rebuilt modules when a new kernel is installed
  - It is included in many distributions

Outline

- Before you start
- Hello world module
- Build, load and unload
- User VS Kernel space programming

User VS. Kernel programming

- kernel module programming
  - similar to event driven programming
- init function
  - says: hey I am here, I will serve your requests from now and on
- exit function
  - says: I am going to leave you.. don't bother trying to find me anymore
- Unload
  - should release any resource that the module had acquired

User VS. Kernel programming

- kernel module runs in kernel space
  - Core of the operating system
  - Privileged operating system functions
  - Full access to all memory and machine hardware
  - Kernel address space
- User programs run in user space
  - It restricts user programs so they can't mess resources owned by other programs or by the OS kernel
  - Limited ability to do bad things like crashing the machine

User VS. Kernel programming

- System calls Switch between user and kernel
- Memory handling
  - malloc is C library call - NOT a system call
    - Use brk system call
  - Kernel allocates virtual memory area for the application
  - Lacks of memory protection
- Portability
  - Kernel modules work with specific version and distribution of the kernel and might be platform-specific

User VS. Kernel programming

- Kernel does not have standard headers
  - Is not linked against the standard C library
  - However, many functions are implemented inside the Linux kernel
- Cannot execute easily floating point operations
  - Floating point operations are architecture dependent
  - Usually, implemented with traps, (trigger integer to floating point mode transition)
  - In the kernel space it requires saving and restoring the floating point operations manually
- Small fixed size stack
  - Configurable at compile time (4KB or 8KB)

Music album as LKM

- Band releases album as Linux kernel module
  - https://github.com/usrbinnc/netcat-cpi-kernel-module